Chapter 8

Planning and Learning with Tabular Methods

In this chapter we develop a unified view of methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model, such as Monte Carlo and temporal-difference methods. We think of the former as *planning* methods and of the latter as *learning* methods. Although there are real differences between these two kinds of methods, there are also great similarities. In particular, the heart of both kinds of methods is the computation of value functions. Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it to update an approximate value function. Earlier in this book we presented Monte Carlo and temporal-difference methods as distinct alternatives, then showed how they can be seamlessly integrated by using eligibility traces such as in TD(λ). Our goal in this chapter is a similar integration of planning and learning methods. Having established these as distinct in earlier chapters, we now explore the extent to which they can be intermixed.

8.1 Models and Planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled

according to the probabilities; these we call *sample models*. For example, consider modeling the sum of a dozen dice. A distribution model would produce all possible sums and their probabilities of occurring, whereas a sample model would produce an individual sum drawn according to this probability distribution. The kind of model assumed in dynamic programming—estimates of the state transition probabilities and expected rewards, $p(s'|s,a)$ and $r(s,a,s')$—is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in surprisingly many applications it is much easier to obtain sample models than distribution models.

Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated experience*.

The word *planning* is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

Within artificial intelligence, there are two distinct approaches to planning according to our definition. In *state-space planning*, which includes the approach we take in this book, planning is viewed primarily as a search through the state space for an optimal policy or path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call *plan-space planning*, planning is instead viewed as a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Plan-space planning includes evolutionary methods and *partial-order planning*, a popular kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning. Plan-space methods are difficult to apply efficiently to the stochastic optimal control problems that are the focus in reinforcement learning, and we do not consider them further (but see Section 15.6 for one application of reinforcement learning within plan-space planning).

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the

learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute their value functions by backup operations applied to simulated experience. This common structure can be diagrammed as follows:

Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value and update the state's estimated value. In this chapter we argue that various other state-space planning methods also fit this structure, with individual methods differing only in the kinds of backups they do, the order in which they do them, and in how long the backed-up information is retained.

Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the estimation of value functions by backup operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases a learning algorithm can be substituted for the key backup step of a planning method. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. Figure 8.1 shows a simple example of a planning method based on one-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample one-step tabular Q-planning*, converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment (each state–action pair must be selected an infinite number of times in Step 1, and α must decrease appropriately over time).

In addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small, incremental steps. This enables planning to be interrupted or redirected at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. More surprisingly, later in this chapter we present evidence that planning in very small

Do forever:
1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(s)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 8.1: Random-sample one-step tabular Q-planning

steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly.

8.2 Integrating Planning, Acting, and Learning

When planning is done on-line, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected in the near future. If decision-making and model-learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an on-line planning agent. Each function appears in Dyna-Q in a simple, almost trivial, form. In subsequent sections we elaborate some of the alternate ways of achieving each function and the trade-offs between them. For now, we seek merely to illustrate the ideas and stimulate your intuition.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model-learning*, and the latter we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values, and policy are summarized in Figure 8.2. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value and policy functions either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.

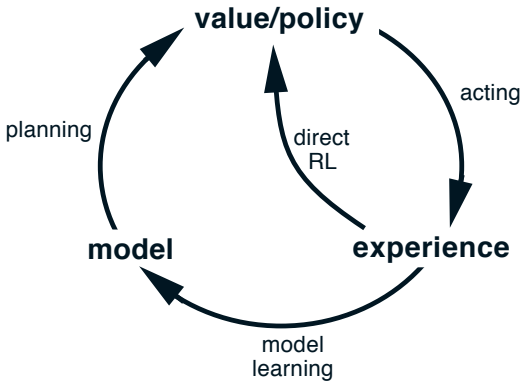

Figure 8.2: Relationships among learning, planning, and acting.

Both direct and indirect methods have advantages and disadvantages. Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model. Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning. Related debates in psychology and AI concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision-making. Our view is that the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for modelfree learning.

Dyna-Q includes all of the processes shown in Figure 8.2—planning, acting, model-learning, and direct RL—all occurring continually. The planning method is the random-sample one-step tabular Q-planning method given in Figure 8.1. The direct RL method is one-step tabular Q-learning. The model-learning method is also table-based and assumes the world is deterministic. After each transition $S_t, A_t \rightsquigarrow R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state–action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction. During planning, the Q-planning algorithm randomly samples only from state–action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is

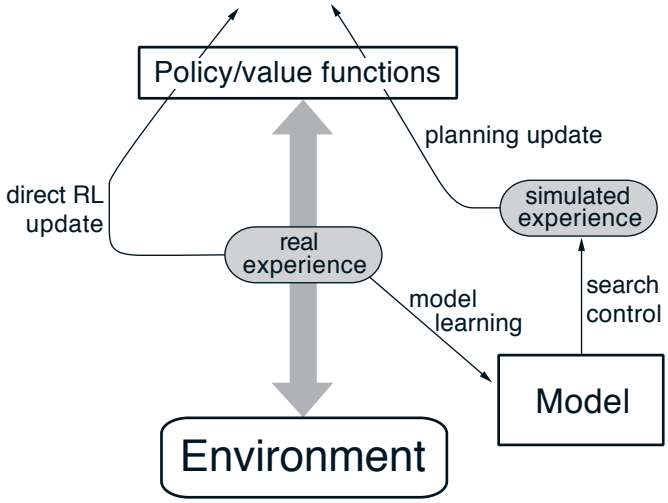

Figure 8.3: The general Dyna Architecture

one example, is shown in Figure 8.3. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the "final common path" for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

Conceptually, planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation-intensive. Let us assume that there is time in each step, after acting, model-learning, and direct RL, to complete n iterations (Steps 1–3) of the Q-planning algorithm. Figure 8.4 shows the complete algorithm for Dyna-Q.

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \epsilon$-greedy(S,Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 8.4: Dyna-Q Algorithm. $Model(s,a)$ denotes the contents of the model (predicted next state and reward) for state–action pair s,a. Direct reinforcement learning, model-learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be one-step tabular Q-learning.

Example 8.1: Dyna Maze Consider the simple maze shown inset in Figure 8.5. In each of the 47 states there are four actions, `up`, `down`, `right`, and `left`, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains where it is. Reward is zero on all transitions, except those into the goal state, on which it is +1. After reaching the goal state (G), the agent returns to the start state (S) to begin a new episode. This is a discounted, episodic task with $\gamma = 0.95$.

The main part of Figure 8.5 shows average learning curves from an experiment in which Dyna-Q agents were applied to the maze task. The initial action values were zero, the step-size parameter was $\alpha = 0.1$, and the exploration parameter was $\epsilon = 0.1$. When selecting greedily among actions, ties were broken randomly. The agents varied in the number of planning steps, n, they performed per real step. For each n, the curves show the number of steps taken by the agent in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of n, and its data are not shown in the figure. After the first episode, performance improved for all values of n, but much more rapidly for larger values. Recall that the $n = 0$ agent is a nonplanning agent, utilizing only direct reinforcement learning (one-step tabular Q-learning). This was by far the slowest agent on this problem, despite the fact that the parameter values (α and ε) were optimized for it. The

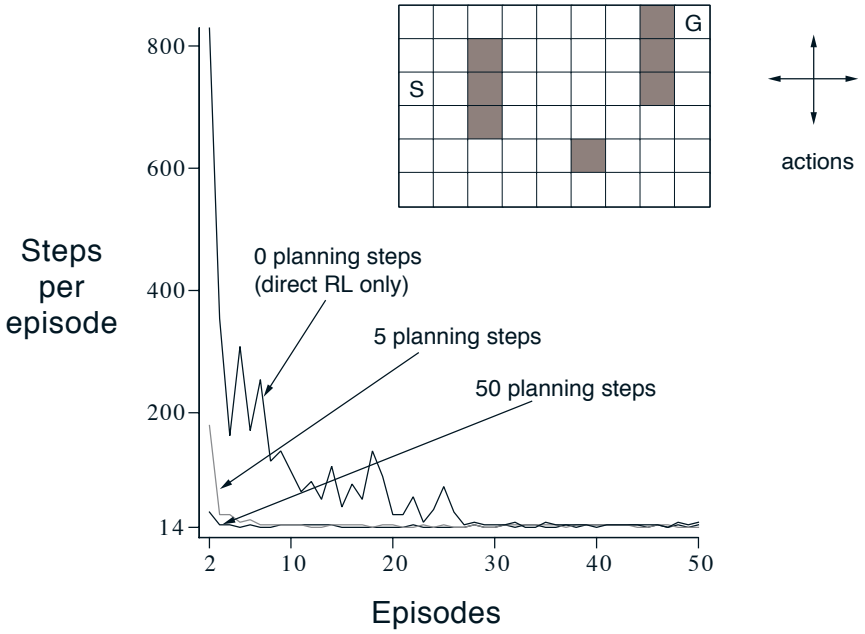

Figure 8.5: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from S to G as quickly as possible.

nonplanning agent took about 25 episodes to reach (ε-)optimal performance, whereas the $n = 5$ agent took about five episodes, and the $n = 50$ agent took only three episodes.

Figure 8.6 shows why the planning agents found the solution so much faster than the nonplanning agent. Shown are the policies found by the $n = 0$ and $n = 50$ agents halfway through the second episode. Without planning ($n = 0$), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the episode's end will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained. ■

In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background. Also ongoing in the

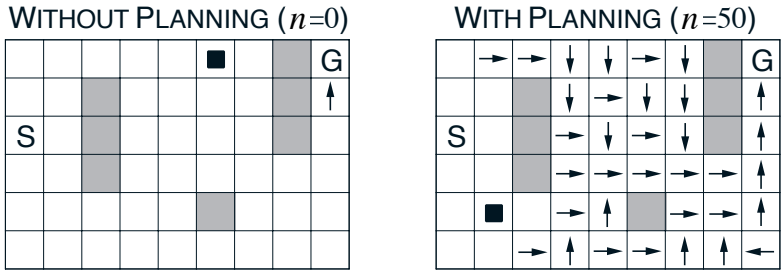

Figure 8.6: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; no arrow is shown for a state if all of its action values are equal. The black square indicates the location of the agent.

background is the model-learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.

8.3 When the Model Is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we cannot expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect, the planning process will compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

Example 8.2: Blocking Maze A maze example illustrating this relatively minor kind of modeling error and recovery from it is shown in Figure 8.7. Initially, there is a short path from start to goal, to the right of the barrier, as shown in the upper left of the figure. After 1000 time steps, the short path is "blocked," and a longer path is opened up along the left-hand side of

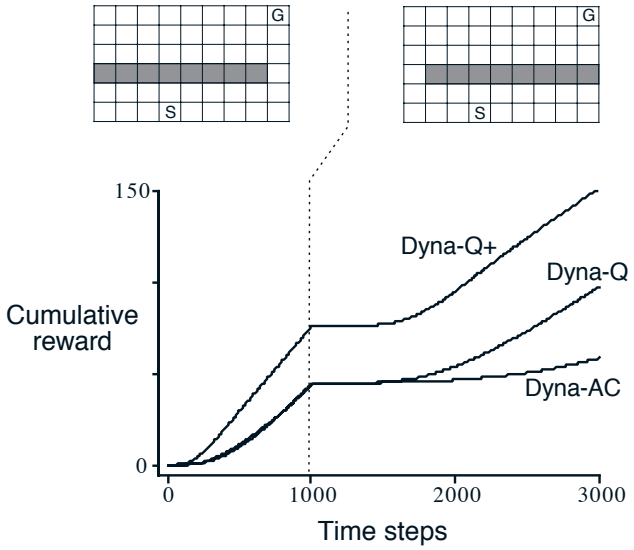

Figure 8.7: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration. Dyna-AC is a Dyna agent that uses an actor–critic learning method instead of Q-learning.

the barrier, as shown in upper right of the figure. The graph shows average cumulative reward for Dyna-Q and two other Dyna agents. The first part of the graph shows that all three Dyna agents found the short path within 1000 steps. When the environment changed, the graphs become flat, indicating a period during which the agents obtained no reward because they were wandering around behind the barrier. After a while, however, they were able to find the new opening and the new optimal behavior. ■

Greater difficulties arise when the environment changes to become *better* than it was before, and yet the formerly correct policy does not reveal the improvement. In these cases the modeling error may not be detected for a long time, if ever, as we see in the next example.

Example 8.3: Shortcut Maze The problem caused by this kind of environmental change is illustrated by the maze example shown in Figure 8.8. Initially, the optimal path is to go around the left side of the barrier (upper left). After 3000 steps, however, a shorter path is opened up along the right side, without disturbing the longer path (upper right). The graph shows that two of the three Dyna agents never switched to the shortcut. In fact, they never realized that it existed. Their models said that there was no shortcut, so the more they planned, the less likely they were to step to the right

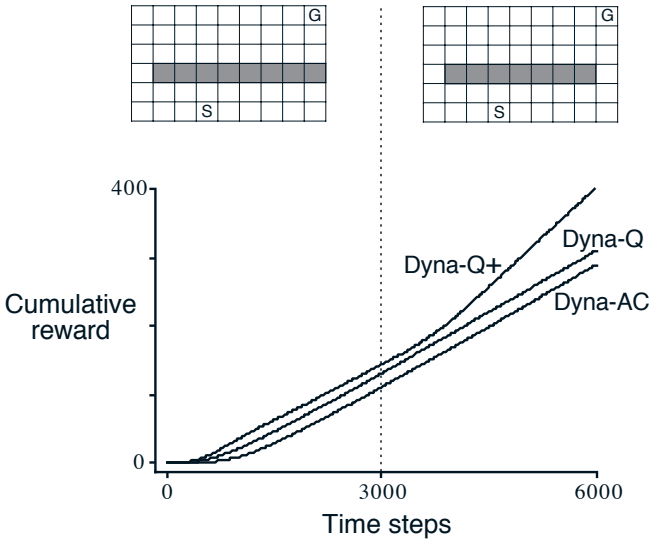

Figure 8.8: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest.

and discover it. Even with an ε-greedy policy, it is very unlikely that an agent will take so many exploratory actions as to discover the shortcut. ■

The general problem here is another version of the conflict between exploration and exploitation. In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model. We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent that did solve the shortcut maze uses one such heuristic. This agent keeps track for each state–action pair of how many time steps have elapsed since the pair was last tried in a real interaction with the environment. The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect. To encourage behavior that tests long-untried actions, a special "bonus reward" is given on simulated experiences involving these actions. In particular, if the modeled reward for a transition is R, and the transition has not been tried in τ time steps, then planning backups are done as if that transition produced a reward of $R + \kappa\sqrt{\tau}$, for some small κ. This encourages the agent to keep testing all accessible state transitions and even to plan

long sequences of actions in order to carry out such tests.[1] Of course all this testing has its cost, but in many cases, as in the shortcut maze, this kind of computational curiosity is well worth the extra exploration.

8.4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions are started in state–action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and backups are focused on particular state–action pairs. For example, consider what happens during the second episode of the first maze task (Figure 8.6). At the beginning of the second episode, only the state–action pair leading directly into the goal has a positive value; the values of all other pairs are still zero. This means that it is pointless to back up along almost all transitions, because they take the agent from one zero-valued state to another, and thus the backups would have no effect. Only a backup along a transition into the state just prior to the goal, or from it into the goal, will change any values. If simulated transitions are generated uniformly, then many wasteful backups will be made before stumbling onto one of the two useful ones. As planning progresses, the region of useful backups grows, but planning is still far less efficient than it would be if focused where it would do the most good. In the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient.

This example suggests that search might be usefully focused by working *backward* from goal states. Of course, we do not really want to use any methods specific to the idea of "goal state." We want methods that work for general reward functions. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states but from any state whose value has changed. Assume that the values are initially correct given the model, as they were in the maze example prior to discovering the goal. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step backups are those of actions that lead directly into the one state whose value has already been changed. If the values of these actions are updated,

[1]The Dyna-Q+ agent was changed in two other ways as well. First, actions that had never before been tried before from a state were allowed to be considered in the planning step (f) of Figure 8.4. Second, the initial model for such actions was that they would lead back to the same state with a reward of zero.